Evolving a Project-based Software Engineering Course: A Case Study

David Delgado[1], Alejandro Velasco[1]
[1]Computer Systems and Industrial Engineering
Universidad Nacional de Colombia
Bogotá, Colombia
dcdelgadoa@unal.edu.co; savelascod@unal.edu.co

Jairo Aponte[1], Andrian Marcus[2]
[2]Department of Computer Science
The University of Texas at Dallas
Richardson, TX, USA
jhapontem@unal.edu.co; amarcus@utdallas.edu

***Abstract*—This paper presents the evolution of a project-based course in Software Engineering for undergraduate students at the Universidad Nacional de Colombia. We describe and explain the changes we have done over six semesters. In addition, we investigate the effects of the changes on the students' grades and their project activities, by analyzing the software project repositories and the student feedback. Most of the changes had positive and expected results, while some had unexpected consequences. We distill a set of lessons regarding the class evolution, which will guide the future improvement of the course and which could be useful for other educators developing a similar course.**

Keywords-project-based learning, capstone project, agile methods, software engineering education

I. Introduction

SWEBOK[1] and the ACM/IEEE Curricula Guidelines[2][3] recommend including team-based projects into the software engineering and computer science curriculum. One of the most common ways implemented by educators across the globe is via a project-based course (*a.k.a.*, capstone or senior project). In some cases, such courses are intimately connected with existing software engineering courses. For example, students start projects in the software engineering class and they finish them in the capstone course. In many other cases, the capstone courses have a software engineering course as pre-requisite and function in complementary fashion.

Regardless of their place in the curriculum, capstone courses face shared challenges, such as, selecting the process to be followed by the teams, selecting relevant application domains, including state of the art tools and techniques for process and team management, etc. For example, many capstone courses focus these days on integrating agile development processes. The application domain often depends on how the projects are defined. Some projects are student-defined, others are instructor-defined, while many more are defined by industry partners. Choosing an application domain that is exciting for students and at the same time relevant for industry is not always easy. For example, web-based and mobile applications are extremely popular with students and industry alike today, but that may change tomorrow. Technical choices are equally important and challenging. Some courses leave choices, such as, programming language, frameworks, and IDE up to the students, while requiring the use of specific repository hosting systems, such as, GitHub. When some of these technologies and practices are not covered by previous courses, then the capstone course must include them as well.

This paper presents the evolution of a software engineering capstone course (*i.e.*, Software Engineering II) underpinned by project-based learning [1], taught at the Universidad Nacional de Colombia (UNAL), sede Bogotá, within the Computer Systems Engineering curriculum. We investigate the evolution of the course over six semesters through the analysis of team projects, student surveys, and instructor notes. Our experience allowed us to distil a set of lessons learned, some of which are echoed in related literature. We expect that they will help further refine the course and, hopefully, will serve as guidance for those trying to include a similar course in their curriculum.

The rest of the paper is organized as follows. Section II describes the course in details, with emphasis on the changes done in the past six semesters. Section III provides an analysis of the effects of the changes, using student evaluation data, student grades, and commit data from the repositories. Section IV presents the lessons learned from our course evolution experience. Section V presents the related work, while Section VI shows our conclusions and future work.

II. Course Description and Evolution

The Software Engineering II (SEII) course described in this paper is required for all undergraduate majors in the Computer Systems Engineering (CSE) program at UNAL. As prerequisite, the students must take the previous software engineering course (Software Engineering I), which introduces software engineering topics, ranging from requirements to testing. They also must have completed traditional programming courses, such as, Introduction to Programming, Object-Oriented Programming, and Data Structures. The course meets formally twice a week and each session lasts two hours. The semester has 16 weeks of classes with a one-week break after the eighth week of classes. Upon completing the SEII course, the CSE students also participate in an *Interdisciplinary Project* course,

[1] https://www.computer.org/web/swebok
[2] https://www.acm.org/education/se2014.pdf
[3] http://www.acm.org/education/CS2013-final-report.pdf

required for all engineering students, where they team up with students from other engineering majors (*e.g.*, electrical engineering, mechanical engineering, etc.) and work on interdisciplinary projects.

The main purpose of the course is to teach students how to carry out a small software development project, while following agile process models and using appropriate tool support for effective teamwork.

We discuss in this Section the evolution of the SEII course over six semesters (between 2014 and 2016). The academic year at UNAL is split into two semesters: Semester I from February to June and Semester II from August to December. For simplicity, we refer to Semester I in 2014 as 2014-I. So, our course evolution narrative spans the following semesters: 2014-I, 2014-II, 2015-I, 2015-II, 2016-I, and 2016-II. During these six semesters, several aspects of the course have changed, such as, process followed by students, technology choices, grading policies, etc. As we discuss these changes, we use visual aids to express how an aspect of the course changed over the six semesters. We use a six-bar icon ▮▯▯▯▯▯ representing the six semesters in chronological order: the left most bar corresponds to 2014-I, whereas the right-most bar corresponds to 2016-II. The empty (white) bars indicates that the described aspect of the course was not covered in its corresponding semester, whereas a filled (blue/gray) bar indicates that the course aspect was covered in its corresponding semester. For example, SEMAT[4] ▮▮▮▮▯▯ indicates that SEMAT was used during the 2014-I, 2014-II, 2015-I, and 2015-II semesters and discontinued after that. Also, Ruby ▯▯▯▯▯▮ indicates that Ruby has been used during the 2016-II semester and it was not used before.

A. Topics Covered

The lecture part of the course covers seven main topics. The extent of each topic has changed slightly over the past semesters, to support other changes in the course. This section describes each topic and its evolution. The approximate length of each one is in brackets.

Agile processes [2-3 weeks]. The instructor covers the principles and core values promoted by the agile movement ▮▮▮▮▮▮. Details of the most popular agile processes are then covered ▮▮▮▮▯▯, such as XP, Kanban, Scrum, etc. Recently, only Scrum[5] ▯▯▯▯▮▮ is covered in details and it is the mandatory process for the teams.

Project set up [1-2 weeks]. During this part of the class, we perform an overview and tutorials of the main tools that the students need to use on their projects. The content varies from semester to semester, based on the main technology choices, as described later in this Section.

Version control [2-3 weeks]. This part of the course reviews the fundamental concepts of version control ranging from setting up a repository to using branches and managing merge conflicts. The goal is to ensure that each student can setup a repository, contribute to it, resolve conflicts, etc. As version control system, we have used Mercurial[6] ▮▮▯▯▯▯ and Git[7]▮▮▮▮▮▮. For hosting each repository, students used Google Code ▮▯▯▯▯▯, Bitbucket[8] ▯▮▯▯▯▯, and GitHub[9] ▯▮▮▮▮▮.

Development frameworks [3 weeks]. In the early editions of the course, the teams were free to work on any type of project ▮▯▯▯▯▯, including video games, kinect apps, mobile applications, and even desktop applications. Starting 2014-II, the projects had to be web applications ▯▮▮▮▮▮, as the instructor and TA could not effectively provide support for all possible technologies and application types.

We have used full-stack MVC frameworks ▮▮▮▮▮▮ because they have all the components pre-integrated into the framework, which significantly simplifies the configuration process, and additionally, they implement the MVC architecture pattern. In this part of the course students learn how the framework operates and how to build the fundamental components involved in web development. Grails[10] ▮▮▮▮▮▯ was the MVC framework used in the class. Grails is based on the Groovy[11] ▮▮▮▮▮▯ progamming language and while it is quite similar to Java, which is part of the students' background, a number of assignments were give to ensure their proficiency with Groovy. Students consistently expressed complaints about the quality and availability of the Grails documentation and support, so it was replaced with Ruby on Rails[12] ▯▯▯▯▯▮. As Ruby is new to the students, it is covered in class. Feedback from the students enrolled in the last recorded semester (2016-II) indicates that the switch was successful; Further analysis during the next semesters will reveal any problems with the switch.

The testing framework [1-2 weeks]. While we used Grails ▮▮▮▮▮▯, students were required to use the JUnit[13] ▮▮▮▮▮▯ and Spock[14] ▮▮▮▮▮▯ testing frameworks. With the introduction of Ruby on Rails, we started using the built-in mechanisms in Rails ▯▯▯▯▯▮ for testing applications at unit, functional, and integration levels.

Object-oriented design [2-3 weeks]. We covered basic properties of object-oriented design (*e.g.*, high cohesion and low coupling), essential design principles (*e.g.*, the open/close principle), and object-oriented design patterns.

Software quality [2 weeks]. Students learn basic concepts of internal software quality and how to use tools like SonarQube[15] ▮▮▮▮▮▯, a software quality management platform, enabling them to assess various quality attributes of their products, ranging from minor styling details to critical design errors. As the main development framework for the projects changed from Grails ▮▮▮▮▮▯ to Rails ▯▯▯▯▯▮, SonarQube was no longer suitable and was removed from the tools used in this section of the course.

[4] http://semat.org/
[5] http://www.scrumguides.org/
[6] https://www.mercurial-scm.org/
[7] https://git-scm.com/
[8] https://bitbucket.org/product
[9] https://pages.github.com/
[10] https://grails.org/
[11] http://www.groovy-lang.org/
[12] http://rubyonrails.org/
[13] http://junit.org/junit4/
[14] http://spockframework.org/
[15] http://www.sonarqube.org/

B. *Team and Project Definition*

Within the first two weeks of classes, the students self-organize in teams of 4 to 6 members. Sometimes members of a team have worked together in previous courses, while others meet for the first time in this course. As soon as the teams are formed, they are encouraged to discuss and make decisions about managerial aspects, such as, deciding who the team leader is, the frequency and type of meetings they will have, and establishing the set of agile values, principles, and practices they will adopt for their project. At the same time, the teams hold brainstorming sessions in which they explore possible projects. The proposed project by a team is often a system in which its members are interested, and therefore, the motivation is usually high along the entire project lifecycle. Sometimes the selected project addresses real needs of a private company or a public institution. In any case, by the third week of classes every group must have an approved project to work on.

C. *Project Management*

Each team was responsible for determining the length of each iteration and establish their own release schedule. We observed that long iterations (*i.e.*, three or more weeks) led to students contributing unequally through the iteration. At the same time, short iterations (*i.e.*, one week or less) posed significant challenges to the instructor and TAs to keep track of each team in a timely fashion. Starting 2015-II, iterations are fixed to two weeks for all teams, starting the fifth week of the semester. All teams have identical release schedules.

The teams can plan and execute each iteration as they see fit, but they had to register and update the progress of each iteration using an appropriate tool. We used Murally[16] and Trello[17], general purpose tools, to perform this iteration control. In the last analyzed semester, we used Taiga[18], an open source project management system for agile projects, which offers Scrum templates and manages backlogs of user stories.

The applications are deployed in a PaaS system. OpenShift[19], Heroku[20] and IBM Bluemix[21] () are PaaS services that provide web hosting and allow users to deploy web applications developed in several languages like Java, PHP, Python, and frameworks like Grails, Rails, Django etc. The teams deploy their application to test it in a production environment, with some architectural limitations.

Teams have the liberty of choosing their favorite communication tool support, in addition to the communication mechanisms offered by the version control system, issue tracker, and project management tools. The TAs and the instructor are involved in the communication channels. Skype, Teamspeak[22], and Slack[23] are the most popular among students.

[16] https://mural.co/
[17] https://trello.com/
[18] https://taiga.io/
[19] https://hub.openshift.com/
[20] https://www.heroku.com/
[21] https://www.ibm.com/cloud-computing/bluemix/
[22] http://www.teamspeak.com/

Likewise, the teams have the freedom to choose their favorite IDE. Eclipse[24], GGTS[25], and IntelliJ IDEA[26] are the most commonly chosen.

In addition, teams are allowed to exchange information between the teams regarding their experience and technical choices. For example, if a team successfully uses a particular library in their project, they are encouraged to share their experience with the other teams, such that they can also use it.

Each project has a product owner, who meets with the team once a week, and as such, influences the order in which the project requirements are met, and eventually, creates, modifies or removes requirements from the project backlog. This role is played by one of the two TAs, when the project does not have an external client.

D. *Deliverables and Grading*

Initially, the grade for the project accounted for 55% of the final grade. The course included assignments, quizzes, exams, lectures, and lab practices, which accounted for the remainder of the grade. With the shift on project-based learning, the weight of the project in the final grade increased gradually to 60%, 70%, and 80%. In the last semesters, the final course grade was computed as follows:

• Assignments	20%
• Project (80%)	
o First project evaluation (week 11)	10%
o Final project evaluation (week 16)	30%
o Individual contributions	40%

The assignments are based on the course topics (*e.g.*, creating and managing repositories, etc.) and are performed in class or at home. The assignments related to the MVC frameworks have evolved the most during the six semesters. During 2014-I and 2014-II, each team had to study several aspects of the framework (*i.e.*, Grails) and then make a presentation in the class. During 2015-I and 2015-II these assignments and presentations were replaced by a set of lectures given by the instructor and TAs and a large assignment aimed to learn how to use Grails in a practical way. The assignment took several weeks to be finished, which impacted the teams' ability to focus on the project definitions. Starting 2016-I, the large assignments were replaced by a set of smaller assignment, which were finished in the class by the students - with few exceptions, when they finished before the next class.

After the teams have been created and projects chose, each team must deliver a project definition document (PDD). Figure 1 shows the structure of the PDD, which follows the organization of the SEMAT () alphas. The PDD is reviewed and approved by the instructor and the TAs. With the adoption of Scrum, as the unique process to be followed by the teams, user stories have been introduced

[23] https://slack.com/
[24] http://www.eclipse.org/
[25] https://spring.io/tools/ggts
[26] https://www.jetbrains.com/idea/

1. *Opportunity*: The set of circumstances that makes it appropriate to develop or change a software system
2. *Stakeholders:* The people, groups, or organizations that affect or are affected by a software system.
3. *Requirements:* A preliminary list of functionalities the system must offer to address the opportunity and satisfy the stakeholders
4. *Software System:* A high level description of the system architecture
5. *Way of Working:* The tailored set of practices and tools used by a team to guide and support their work
6. *Work:* Optionally, the team could describe a plan for the first iteration of the project
7. *Team:* brief descriptions of the group of people engaged in the development of the proposed software system

Figure 1. The structure of the PDD document.

to describe the Requirements alpha, as SEMAT (▥) was phased out due to immature tool support.

At the end of each iteration, each team member receives a grade for their individual work ▥, per the tasks assigned to her in the iteration plan and her contributions to the project repository (*i.e.*, commits). This is a laborious activity for the graders, which inspect every commit. The introduction of two TAs in 2016-I greatly facilitated this practice and allowed the transition to the 80% ▥ grade weight for the project.

Each team makes two formal presentations of their project, during week 11 and week 16, respectively. Each team has about 25 minutes to show the work performed so far. One way of doing that is showing the plan and outcomes of each of the iterations performed and describing the produced software artifacts. Each presentation ends with a demo of the latest stable version of the system to show the implemented functionality.

In addition, at the end of the semester all groups participate in an open exhibition where they present their projects to the public and the School community. This presentation is not graded.

III. Projects and Student Feedback Analysis

The Computer Systems Engineering program at UNAL is not accredited by any independent accreditation board, such as, ABET[27]. The Department collects student evaluations every semester, geared primarily to support faculty and TA evaluations. There is no formal, department-level, course improvement process. As such, individual faculty who teach classes over a longer period, often undertake ad-hoc processes to improve the courses, based on student feedback, personal observations, and less frequently based on measurements of student outcomes. This case study is an example of opportunistic course evolution, with positive results.

Many of the changes from the last six semesters, reported in Section II, are motivated by observations of the instructor and the TAs and student feedback. The main goal of the changes is to focus the course more on project-based learning than in its earlier incarnations. An important sub-goal was to improve the ability of the instructor and TA to get involved more in each project and provide better and more frequent feedback. As such, many of the changes resulted in restricting student choices.

We investigate in this Section data about the student projects and their work, as well as formal student surveys we conducted.

A. *Project Data Analysis*

While many of the course changes were implemented starting 2014-I, we have detailed project and effort data available for three semesters: 2015-I, 2015-II, and 2016-I. We will focus our analysis on the data collected during these semesters from 19 teams, totaling 88 students. In the last analyzed semester (2016-II), six more teams participated in the course (30 students). We omit the data from the last semester (2016-II) because we made two major changes in that semester (*i.e.*, the web framework and the language) and we need data from more semesters before we can properly analyze that change.

1) Student grades

An important goal throughout the course evolution was to make sure the students' success rate in the class improves, especially as the weight of the project increased from semester to semester. Figure 2 shows box plots of the students' grades over three semesters, when the project grade weight increased from 60% ▥ to 70% ▥ and then to 80% ▥.

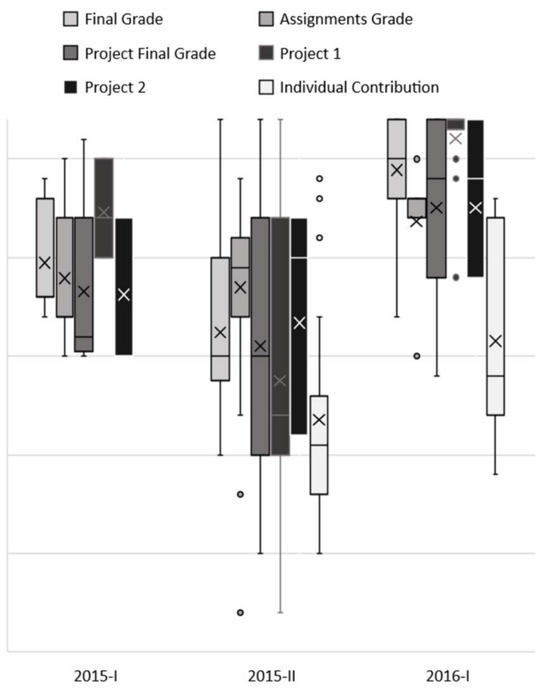

Figure 2. Grades per semester. The 2015-I semester does not have the *individual contribution* component in the project grade.

[27] http://www.abet.org/

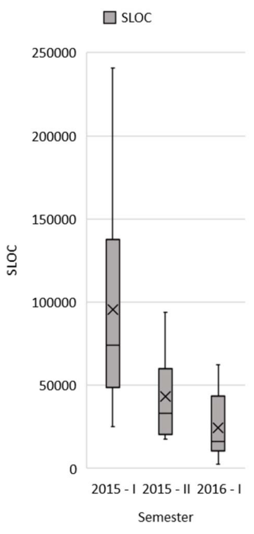

Figure 3. Average project size in SLOC

We analyze the grades in more details. Figure 2 reveals that the introduction of the *individual contribution* grade in 2015-II had a negative effect on the grades of the students. Many of the changes in the course (*e.g.*, uniform use of GitHub and iterations) were introduced primarily to allow us to account for individual project contributions. Historically, every semester several students complained that some of their team members contribute less than others. The grades in 2015-II reflect that fact. The change in grading policy spread quickly from word to mouth among the UNAL students (a common phenomenon) and the next semester saw an increased effort from the students, as highlighted by the individual contribution grade in 2016-I. Collected data from the last analyzed semester (*i.e.*, 2016-II) confirms that the students continue to improve their individual contributions.

Another interesting fact is the sharp increase in the *First project presentation* grade. We attribute this change to the fact that we replaced the large Grails assignment , which students worked on at home, with the smaller assignments that they typically finished in class, giving them more time at home to work on the project proposal and early iterations. We attribute the increase in the assignment grade to the same change. We also think that the change in the assignment structure and how the MVC framework material was covered led to a better understanding of the technology by most students and, consequently, in better projects, as reflected by the grades and the complexity we discuss in the following subsection.

2) Project complexity and deployment

Starting 2014-II the students were asked to define projects that were web applications only . The material covered in class on the relevant frameworks and the corresponding assignments changed and we noticed a much better use of the frameworks by the students. In consequence, the size of their code, in terms of SLOC (Source Lines of Code) – measured with CLOC[28], decreased over time. Figure 3 reflects the trend.

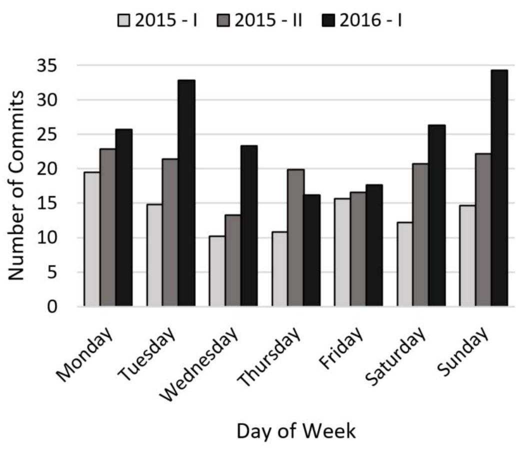

Figure 4. Average commits per day, per project.

The data also indicates that the amount of coding effort in the projects is significant. More importantly, the instructor and TAs note that the projects involved more complex features from semester to semester and that the final version of each project was more mature from semester to semester. We believe that this trend is in part due to the focus of the class on a reduced number of technologies and more time dedicate mastering them by the students. We did not track the time the students spend on the assignment outside the project, but we plan to track it in the future.

Table 1 indicates that more demonstrable projects were deployed in the last two semesters than before, in fact achieving 100% deployment rate in 2016-I. Furthermore, during the last semester (*i.e.*, 2016-II – not included), all teams deployed on Heroku by the 11th week of classes, which confirms that the trend continues.

TABLE 1. NUMBER OF DEPLOYED PROJECTS

Semester	Heroku	Openshift	Groups	Deployment Percentage
2015-I	2	1	6	50.0%
2015-II	4	0	7	57.1%
2016-I	6	0	6	100.0%
Total	**13**	**3**	**31**	**51.6%**

3) Commit data

We analyze the commit activities of the students to determine if the changes related to the length of the iteration affected significantly the commit behavior. We focus the analysis on the time of the commits and investigate fine grained time intervals, such as, time of the day (see Figure 5), and day of the week (see Figure 4). We also analyze commits per iteration (see Figure 7) and per week of iteration (see Figure 6 and Figure 8).

[28] http://cloc.sourceforge.net/

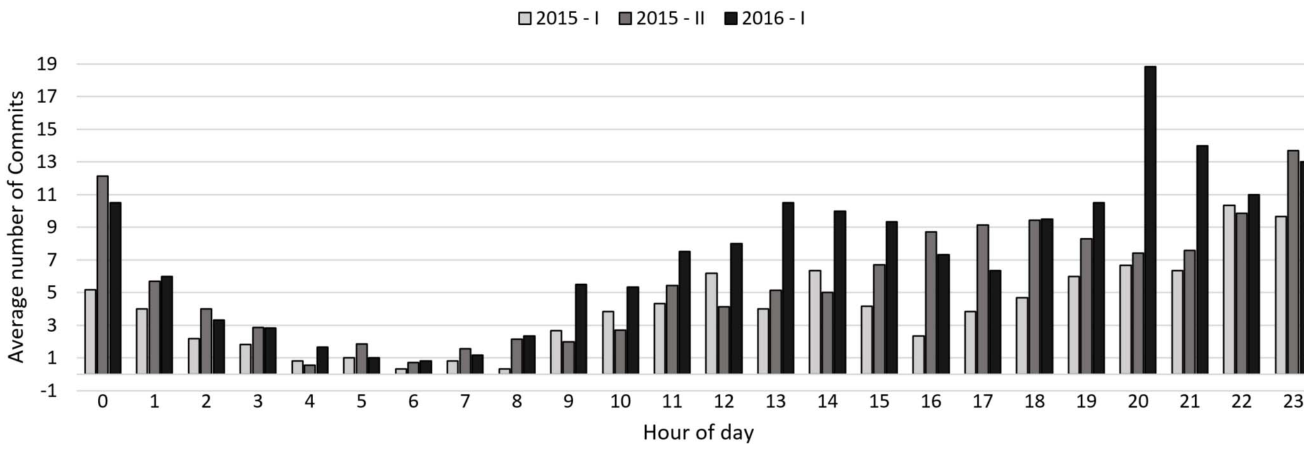

Figure 5. Average commits per hour, per project.

During these three semesters (*i.e.*, 2015-I, 2015-II, and 2016-I) the classes were held on Tuesdays and Thursdays between 2 pm and 4 pm. During 2015-I, the iteration length and deadline was not uniform across teams. During 2015-II and 2016-I, each iteration was two weeks long, starting on Mondays and ending on Sundays. We expected that the days prior to the deadlines to see many commits, compared to the other days of the weeks. As Figure 4 shows, our expectations were not met, the data indicates that the students tend to make more commits on Sundays, Saturdays, Mondays, and Tuesdays, than on Wednesday, Thursdays, and Fridays. Given the data we have we conclude that the recent changes did not impact the weekly work habits of students.

We also analyze the times of the day when the students performed the commits (see Figure 5). As the data shows, we found that students perform commits at all hours of the day. However, the most 'productive' hours are from 7 pm until 1 am. It indicates that the UNAL student population prefers to work on such projects in the evenings. With that in mind, we plan to have all assignment and iteration deadlines set for morning times.

The more important analysis is focused on the introduction of the fixed two-week iteration in 2015-II. The rationale, as explained in the previous section, was based on the fact that in previous semesters, when the iterations were longer, the students had the tendency to postpone work to the end of the iteration and/or shift the iteration length. Short iterations caused delay in feedback from the instructor and the TAs.

Figure 6, Figure 7, and Figure 8 show commit data related to the two-week iterations. Figure 7 indicates that the number of commits increases with each iteration through the semester.

To assess whether the students still tend to postpone the commits to the end of the iteration we compared the commits between the two weeks of the iterations.

Figure 6 shows that the numbers of commits in the second (*i.e.*, the last) week of the iterations is not substantially larger (in average) than in the first week of the iterations. We analyze in more details the aggregate data from Figure 6. Figure 7 shows the average number of commits per project for each week of every iteration, during the two semesters. The detailed analysis shows that the number of commits are sometimes higher in the first week of the iterations, whereas in other iterations the students commit more in the second week. All in all, we conclude that the two

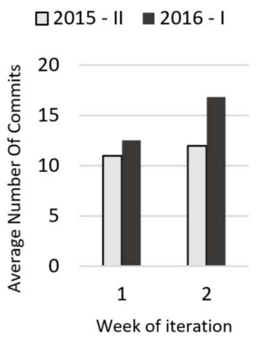

Figure 6. Average number of commits per week of iterations, per project. Iterations are set to two weeks in these semesters.

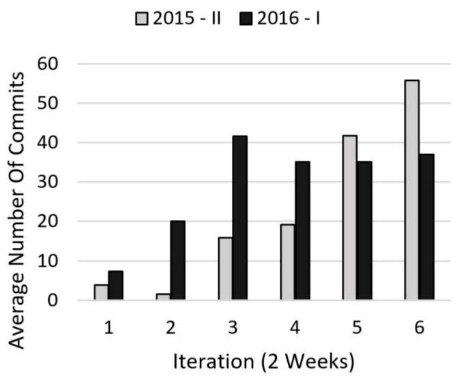

Figure 7. Average number of commits per iterations, per project. Iterations are fixed to two weeks in these semesters.

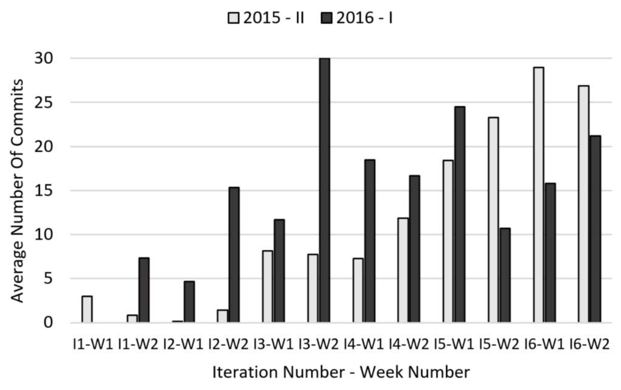

Figure 8. Average number of commits per iteration week, per project. Iterations are fixed to two weeks in these semesters.

week iteration length allows students to distribute the work evenly between iterations, and in that way, keep a sustainable pace throughout the entire project span.

Figure 7 and Figure 8 also show that in 2016-I, the students made more commits in the first few iterations than in 2015-II. We attribute this change to the weekly meetings held between the team and the product owner ▯▯▯▮, in which the participants review the planned tasks' advance and the team and technical difficulties that may have appeared.

4) Software quality

SonarQube ▮▮▮▯, was used by the teams for several semesters to evaluate the quality of their Groovy code. It was mostly used to assess the quality of the final releases of the projects, and to address the most critical issues reported by this tool. As more and more of the code produced by the teams was no longer in Java or Groovy, we stopped using SonarQube in 2016-II. We also used SonarQube to analyze the final projects from the last semesters (see Figure 9). The high number of minor issues is not entirely surprising. Minor issues are defined as "quality flaws which can slightly impact the developer productivity", such as, lines should not be too long, *switch* statements should have at least three cases, etc. As a rule, students were less inclined to fix these types of issues, especially at the end of the semester.

Unexpected was the sharp increase of the major issues. Major issues are defined as "quality flaws which can highly impact the developer productivity", such as, uncovered piece of code, duplicated blocks, unused parameters, etc. Despite the SonarQube terminology, many of the major issues do not have a major impact on the developer productivity; hence, they are often perfect candidates for technical debt generation. Note that these issues do not correlate with external quality attributes. In fact, we observed that the projects in the later semesters had fewer functional errors and the feature sets were more complete (see also the deployment analysis discussed before). We believe that the use of SonarQube through the semester lead to less technical debt. Conversely, less time spent on removing technical debt issues, meant potentially more times spent on adding new features or fixing bugs. As the focus of the projects was on producing complete applications, we accepted the internal

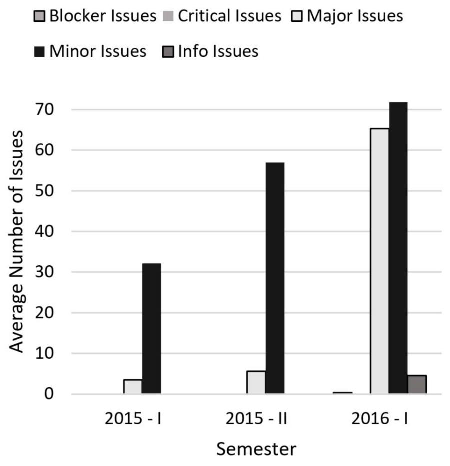

Figure 9. Average number of issues flagged by SonarQube per project.

quality price (as revealed by SonarQube). It is hard to balance both aspects, given all the constraints. We need to consider a better trade-off between external and internal quality in the future.

B. *Students' Feedback*

At the end of each of the last three Grails-based semesters (*i.e.*, 2015-I, 2015-II, and 2016-I), the students were asked to participate in a survey to assess their perception of the agile practices they used through the semester. The participation was optional and 62 of the 90 students responded.

As mentioned before, in the early semesters, XP+SEMAT ▮▮▮▯ was the dominant process used by the teams, whereas later, Scrum ▯▯▯▮ was used by all teams. In all cases, the teams did not follow the processes strictly, but rather they chose the principles and rules they prefer to follow. The rationale is that some of the principles cannot be monitored or enforced, such as, daily scrum meeting, customer on site, etc.

The survey has 13 questions aimed to capture the students' opinion on the agile practices. For each practice, the students could indicate whether they used that practice during the semester or not. In case they have used it, they were asked to assess on a 5-point Likert scale whether the practice had a positive or negative impact on their work: strongly negative impact, negative impact, no impact, positive impact, strongly positive impact. Table 2 summarizes the answer of the students.

The highlighted cells (green) indicate that more than a half of students reported a (strongly) positive impact in using the corresponding practices in their project: incremental and iterative development, collective ownership, fixed length iterations, continuous delivery, continuous integration and periodic meetings, simple design, and pair programming. The least common practices turned out to be: planning game, unit testing, and code refactoring. The most polarizing practices proved to be: pair programming and periodic meetings.

Students were also requested to indicate the main problems they had to face during the course. The more significant problems they reported were associated with the task estimation times, work distribution, communication between team members, and task planning.

In the last part of the survey, students were requested to list aspects of the course they did not like. The large assignments were listed as a major negative aspect of the course, as the students felt their time was better spent on the main projects. Later changes (as described above) confirmed this hypothesis.

TABLE 2. STUDENT ANSWERS TO THE SURVEY ON AGILE PRACTICES. 62 OF 90 STUDENTS RESPONDED.

Question	Not used	S. neg.	Neg.	Neutr.	Pos.	S. pos.
Incremental Development	9	0	0	13	37	3
Iterative Development	4	0	0	8	42	8
Collective Ownership	9	0	2	13	33	5
Code Refactoring	20	1	2	14	21	4
Fixed Length Iterations	10	0	1	13	29	9
Unit Testing	20	0	0	20	19	3
Continuous Delivery	12	0	1	13	27	9
Continuous Integration	10	1	4	7	28	12
Periodic Meetings	2	2	0	4	33	21
Pair Programming	13	4	2	7	21	15
Simple Design	14	1	0	11	29	7
User Stories	12	0	3	6	24	17
Planning Game	45	0	0	8	8	1

Students also pointed out that currently Ruby on Rails is more popular than Groovy and Grails. Their comments were the main reason of the switch to Ruby on Rails in 2016-II.

IV. LESSONS LEARNED

We distill in this Section a set of lessons we learned from our experience. Some of them are supported by the data we collected and the analyses presented in the previous Section. Others reflect informal observations of the instructor and TAs, based on their interactions with the students through the semesters. At the same time, some of these lessons may be applicable in other similar settings, whereas others may not. We do not intend for these lessons to be prescriptive, but rather descriptive of our experience.

1) Use of agile processes

The academic environment does not allow a strict implementation of specific agile processes (*e.g.*, Scrum or XP). The instructors, in consultations with the teams, should be determining which practices and principles to follow. While students prefer some agile practices and principles to others, we found the following to be most helpful:

- Working software is the principal measure of progress. Applying this principle in a strict way allowed more projects to be deployed, resulting in demonstrable applications at the end of the semester. In 2014-II only one team implemented enough functionality to consider the system as demonstrable for the clients. In 2015-II two groups reached that status. In 2016-I four projects reached that status. In 2016-II all projects implemented key functionality and user interfaces were good enough to consider the systems as usable. However, investigation of student code indicates that the pressure to obtain more usable applications comes at the expense of lower internal code quality. Thus, continuous quality assessment practices should be included in the software processes followed by the teams.
- Deliver working software frequently. We found a fixed length iteration of two weeks, imposed on all teams offers the best balance between student and grading effort. The commitment to implement new functionality every two weeks and the feedback received on each delivery promote a more sustainable development effort among the teams, so they are prevented from making heroic efforts at the end of the semester to try to save the project. The behavior of the commits per iteration supports this observation, as in the last semester the commits are more evenly distributed per iteration than before.
- The instructor and/or the TAs should play a key role in each team (in our case as product owner) and participate in weekly meetings. We credit this change with the fact that most recent teams managed to produce more complete applications by the end of the semester.

In earlier work, Zorzo et al. [2] also observed that there is a need to use a modified version of Scrum in academic settings. Muller and Tichy [3] pointed out that it is unclear how to reap the potential benefits of pair programming, although pair programming produces high quality code. The use of pair programming in our course was polarizing among students and its benefits unclear.

2) Use of management tools

The use of common project management tools across all teams allow for better monitoring and grading, which in term improves student activities. We found Trello and GitHub to be especially helpful in allowing us to monitor and grade individual student effort, which lead to better student performance. Problems in teams were detected and corrected early. Rajlich [4] also noted in previous work that defining a mechanism to provide individual grades in team projects is essential to maintain fairness.

3) Student team size and interactions

Given the profile of our students (*i.e.*, academic background, other courses they enroll in in parallel, etc.), we found that groups of five students (no less than four and no more than six) help achieve best the goals of the course. Teams with fewer than four members are unlikely to generate the dynamics and issues that are common on collaborative software endeavors. Also, smaller groups were unable to complete substantial projects in the allotted time. Larger groups faced other kind of problems, such as, inability to meet all together and many other coordination issues. Two large groups in the past performed poorly, whereas another, more recent one, was split earlier in the semester and the new smaller groups performed better.

All teams should have identical timelines and iteration schedules. This practice improves monitoring and grading and allow the definition of projects of similar complexity. From the students' point of view, this practice is positive because all teams are better informed about the progress of the other projects, which promotes a sense of competence throughout the iterations, and encourages the sharing of technical knowledge among teams. On the other hand, we have noticed that when a team has control over the start, end, and duration of the iteration, they tend not to be strict with those limits, such that the iteration length often is shifted, breaking a fundamental rule of agile management.

4) *Technology choices*

There are many technology choices to be made. We advocate dividing them in two categories: mandatory – students must use these technologies/tools (in our case these include now Ruby on Rails, GitHub, Taiga); optional – students can choose from a list of options (in our case the IDEs options include IntelliJ IDEA, Eclipse, and Netbeans; and the PaaS alternatives include Heroku, OpenShift, and IBM BlueMix). The mandatory technologies should be covered in class as early as possible, to allow teams to start the project developments as soon as possible. The Instructor and TAs should be able to provide support with the optional technologies. Students can choose additional technologies at their own risk, considering they may not be able to get support from the instructor and TAs.

5) *Project topics*

We have learned that is better to allow teams to select their own project than giving them a list of possible projects or impose a specific one. This lesson is based on our observations of the students' attitude through the semester. We have noticed that when the team selects and defines the projects, their level of commitment and excitement to the project rises as the software system grows. At the end of the semester the students have a strong sense of ownership towards the project, rather than feeling that they have just done one additional assignment. Most of them are proud of the software product they built, and in some cases, they continue working on the system after they finish the semester, or they consider the system as a product that is worth putting in their professional portfolios. We noticed that, in such situations, the students are more willing to learn technical topics on their own and search for specific tools and frameworks that may help them to build the software system. The obvious side effect of such a strategy is that it may be more difficult to ensure that the projects are of similar complexity. However, previously mentioned practices should allow adjusting more frequently through the semester. We need formal surveys with students in the future to confirm our informal observations regarding this last lesson.

6) *Course organization*

While our collected data did not allow for a fine-grained analysis of the effect of all aspects of the course on student performance and satisfaction, we distill some aspects of the course organization that were informally appreciated by the students and TAs. The project should have the highest weight in the final course grade, and part of that grade should evaluate the individual contributions of each student. Thus, the students must know from the beginning that working on the project is the key factor to success in the course.

The technical topics included in the syllabus should support the development of the project directly and be covered early in the semester.

When it is not possible to have an industry client, the role of the product owner should be performed by someone external to the team, ideally a TA or the course instructor. The product owner should meet the team periodically and help them to plan the iteration tasks and solve organizational problems.

V. Related Work

Educators have proposed different approaches to improve the teaching methodology in software engineering courses. For example, Francese et al. [5] applied a related methodology to the one presented in this paper during the evolution of the mobile application development course at the University of Salerno. The authors implemented a teaching strategy founded on the principles of project-based learning, where software projects are developed by students organized in teams. Lee et al. [6] adopted a course methodology which is supported by using a software design studios. In their approach, the practice and hands-on work are emphasized over other aspects.

The use of agile methodologies in software engineering course has also been widely adopted. Zorzo et al. [2] suggest to use agile methodologies like Scrum in order to teach students how to effectively manage software projects, keeping a balance between the theory and the industry needs. During the course conducted by the authors, students had to work in a project applying the Scrum practices that were explained by the instructors, following an iterative model of eight sprints with three months each. Shukla and Williams [7] adopted practices of extreme programming as main methodology to teach a software engineering course. In this course, the students' perceptions of each principle of extreme programming were evaluated by the authors. During the course, the students worked on four projects and were organized in teams of four people without supervision. Muller and Tichy [3] also adopted extreme programming as software development methodology. As in previous work, the authors evaluated the student opinions for each principle of the methodology.

Rajlich [4] describes a set of "Deadly sins" that were tried to avoid during the evolution of a software engineering course. Razmov [8] emphasizes the relevance of feedback as a fundamental part in every software development process. The author proposed a teaching model following the process of "Doing", "Reflecting", and "Feedback". In a similar way, Roach [9] defends the notion retrospective processes obtained during the course execution and the project development. Jarzabek [10] presents a particular methodology based on APIs, which takes advantage of the interoperability properties.

Our paper adds to this body of knowledge, while reinforcing some of the conclusions of previous experiences.

VI. Conclusions and Future Work

We have been developing a software engineering project-based course at the Universidad Nacional de Colombia, over the past six semesters. The development process is somewhat ad-hoc and opportunistic, in the absence of institutionally sanctioned course improvement methodology.

Over the six semesters, the course underwent many changes, with the main goal of making the project the main component of the course. Changes ranged from the topics covered in class and grading, to the process and technologies used by the teams. The most successful changes were those that allowed the instructor and TAs to take a more active role in each team and better monitor and grade the student activities. Specifically, imposing common iterations schedules and technology choices on the teams and designating the instructor or the TAs as project owners the teams. Among the benefits, students improved their grades and the functionalities of their applications. Some changes led to uneasy trade-offs. For example, allocating more time for improving the features of the applications, at the expense of close monitoring of internal code quality, resulted in an increase in technical debt.

Our experience allowed us to distil a set of lessons learned, some of which are echoed in related literature. We expect that at least some of them will be useful for anyone implementing a similar course.

Looking forward, we need to conduct future student surveys to confirm some of our lessons that are derived from informal observations. In addition, we plan to formalize the course improvement process by defining specific student outcomes and measurements, akin to courses that are part of accredited programs.

References

[1] P. C. Blumenfeld, E. Soloway, R. W. Marx, J. S. Krajcik, M. Guzdial, and A. Palincsar, “Motivating Project-Based Learning: Sustaining the Doing, Supporting the Learning,” *Educ. Psychol.*, vol. 26, no. 3–4, pp. 369–398, Jun. 1991.

[2] S. D. Zorzo, L. de Ponte and D. Lucrédio, "Using scrum to teach software engineering: A case study," 2013 IEEE Frontiers in Education Conference (FIE), Oklahoma City, OK, 2013, pp. 455-461.

[3] M. M. Müller and W. F. Tichy, “Case Study: Extreme Programming in a University Environment,” in *Proceedings of the 23rd International Conference on Software Engineering*, Washington, DC, USA, 2001, pp. 537–544.

[4] V. Rajlich, “Teaching Developer Skills in the First Software Engineering Course,” in *Proceedings of the 2013 International Conference on Software Engineering*, Piscataway, NJ, USA, 2013, pp. 1109–1116.

[5] R. Francese, C. Gravino, M. Risi, G. Scanniello, and G. Tortora, “On the Experience of Using Git-Hub in the Context of an Academic Course for the Development of Apps for Smart Devices,” presented at the The 21st International Conference on Distributed Multimedia Systems, 2015, pp. 292–299.

[6] J. Lee, G. Kotonya, J. Whittle, and C. Bull, “Software Design Studio: A Practical Example,” presented at the Proceedings of the 37th International Conference on Software Engineering - Volume 2 Pages 389-397, 2015, pp. 389–397.

[7] A. Shukla and L. Williams, “Adapting extreme programming for a core software engineering course,” presented at the Proceedings of the 15th Conference on Software Engineering Education and Training (CSEET’02), 2002, pp. 184–191.

[8] V. Razmov, “Effective pedagogical principles and practices in teaching software engineering through projects,” presented at the Proc. 37th FIE, 2007, p. S4E–21–S4E–26.

[9] S. Roach, “Retrospectives in a software engineering project course: Getting students to get the most from a project experience,” presented at the 2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T), 2011, pp. 467–471.

[10] S. Jarzabek, “Teaching advanced software design in team-based project course,” presented at the Software Engineering Education and Training (CSEE&T), 2013 IEEE 26th Conference, 2013, pp. 31–40.